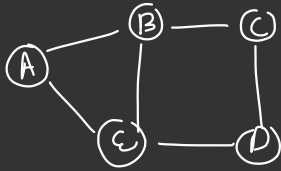




# [ GRAPHS - II ]



Storage

① Edge List

A, B, C, D, E

{ AB, BE, ... }

LL Arraylist



③ Adj List [ array of list, hashmap<string, list<string> ]

A → (B, E)

B → (C, E) A

C → (B, D)

⋮

② Adj Mat

	A	B	C	D	E
A					
B					
C					
D					

bool / int

Algo to Traverso the graph

itr → ①

BFS

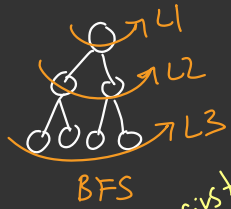
→ application: SSSP on unweighted graphs

rec → ②

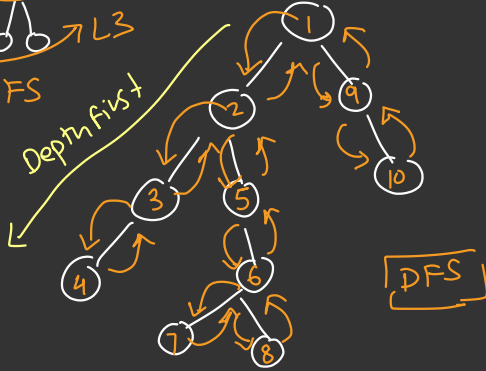
DFS

# Depth First Search

- Rec way of traversing the graph



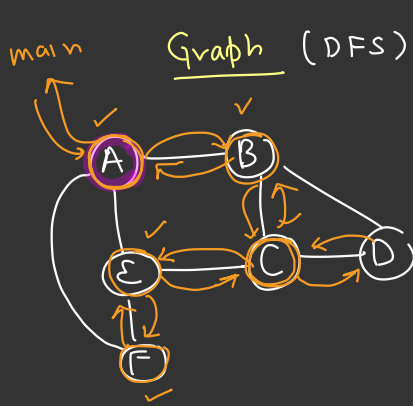
Trees



Preorder (DFS)

- Root
- Left
- Right

DFS



adj list

A → (B, ~~E~~, ~~F~~)

B → (~~A~~, ~~C~~, ~~D~~)

C → (~~E~~, B, D)

D → (~~B~~, ~~C~~)

E → (~~A~~, ~~C~~, ~~F~~)

F → (~~E~~, ~~A~~)

Node  
↓  
Children

A, B, C, E, F, D

- maintain a visited array to stop visiting same node again

Wrapper Fn

dfs ( list<int> adjList[], int v ) {

v=4  
{ 1, 2, 3 }

// default value is  
False, otherwise  
set it to  
False

only  
once

=> bool visited[] = new bool[v]

dfsHelper ( adjList, visited, 0 )

logic  
void

traverses one component

dfsHelper ( list<int> adjList[], bool visited[], int node ) {

print(node)

visited[node] = True

for ( nbr : adjList[node] ) {

if ( visited[nbr] == false ) {

dfsHelper ( adjList, visited, nbr )

}

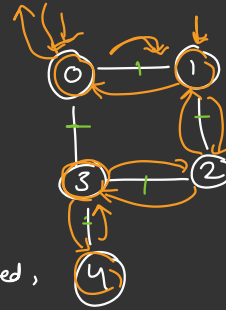
}

return ( optional )

called  
once for  
each  
node

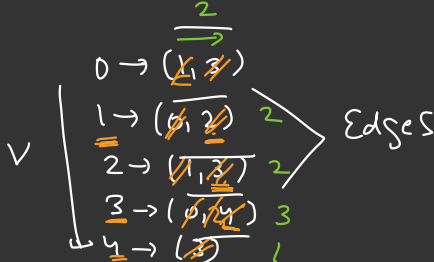
$O(V + \underline{\underline{2E}})$

$= O(V + E)$



SC  
visited + callStack  
 $O(V)$

DFS  
TC:  $O(V + E)$



✓✓ class graph {

list<int> adj list[ ]

→ no need to pass as parameter

→ shared across all the fns

==

bfs()

...

dfs()

...

}

list<int> adj list[ ]

bfs( —, —, →

dfs(                    )

Assuming  $N$  nodes  $\{0, 1, \dots, N-1\}$

Q.

$N$  nodes

$\{A, B, C, D, E, \dots, X, \text{Delhi}, \dots\}$   
0 1 2 3

if nodes are not sequence

✓  
hashmap <string, bool>

visited;

PROBLEM.  $N=9$ , find no of connected components in the graph.

Input

$0 \rightarrow 1, 2$

$1 \rightarrow 0, 3$

$2 \rightarrow 0, 3$

$3 \rightarrow 1, 2$

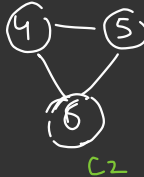
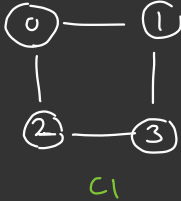
$4 \rightarrow 5, 6$

$5 \rightarrow 4, 6$

$6 \rightarrow 4, 5$

$7 \rightarrow 8$

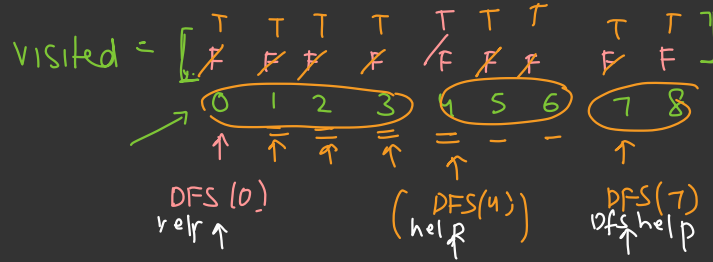
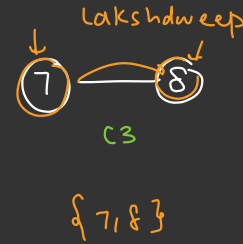
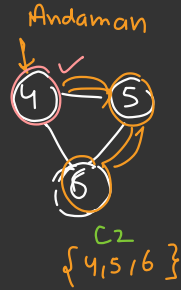
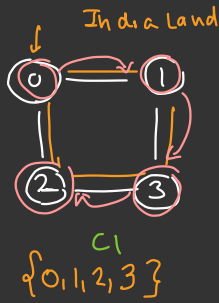
create it  
 $O(V+E)$



= 3 components

A component is said to be connected if from every node we can visit all the nodes inside the component.

one graph



$$C1 = 4$$

$$C2 = 3$$

$$C3 = 2$$

List<int>

int dfs ( adjList[], int v ) {

bool visited[] = new bool[V]

count = 0

init(visited, false)

for ( i=0 ; i<V ; i++ ) {

if ( !visited[i] ) {

⇒ dfs Helper ( adjList, visited, i ) → Print all nodes in the component

count++

return count;

}

OR it can BFS start from i

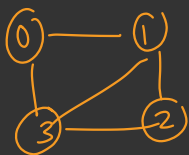
TC :  $O(V+E)$  for BFS  
SC :  $O(V+E)$  & DFS.

Input → edge List

↓

adjList ( $O(V+E)$ )

Connected Components



4 5

0 1

1 2

2 3

3 0

1 3

x y

Graph:  $g;$

$N = \text{input}()$

$E = \text{input}()$

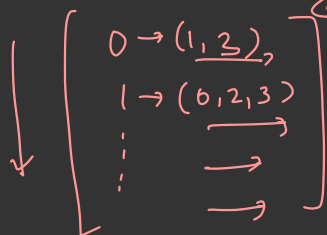
for ( $i=0; i < E; i++$ ) {

    Read  $x, y$

$g$  addEdge( $x, y$ )

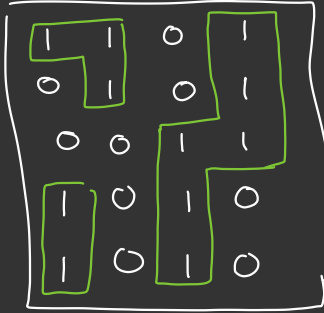
3

$O(V+E)$  space



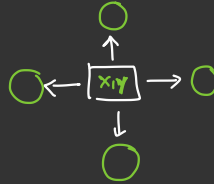
$O(V+E)$

## 2D Matrix



↓  
⇒ 3 islands.

0 - water  
1 - land

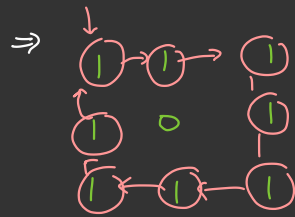
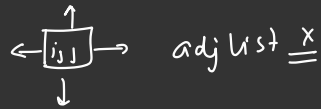


# No of islands

TRY it  
10 20

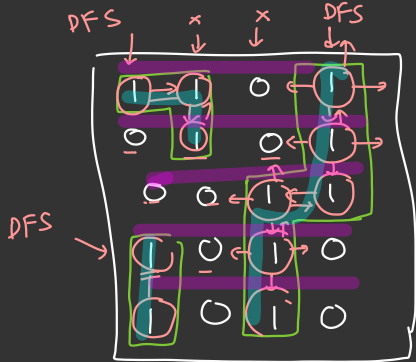
int getIslands (mat[ ][ ], N, M)

→ 0 ⇒ nhrs



cycle ?

visited[ ][ ]



Implicit Graph

3 calls

dfs Helper (mat[7][7], vis[7][7], int i, int j) {

if (mat[i][j] == 0 OR i, j is outside mat OR visited[i][j] == True)

return

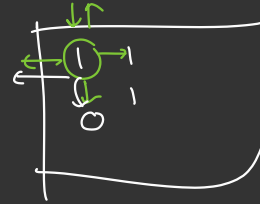
vis[i][j] = True

dfs Helper (mat, vis, i, j-1)

- dfs Helper (mat, vis, i, j+1)

- dfs Helper (mat, vis, i+1, j)

- dfs Helper (mat, vis, i-1, j)



Rec on  
2D  
array

}

getIslands → dfs (mat[i][j], N, M) {

bool visited[M][N] = {false}

// create and set it false



count = 0

for (i = 0 \_\_\_\_\_ M-1)

for (j = 0 \_\_\_\_\_ N-1) {

if (mat[i][j] == 1 && !visited[i][j]) {

dfsHelper(mat, vis, i, j)

count++

}

}

}

return count,

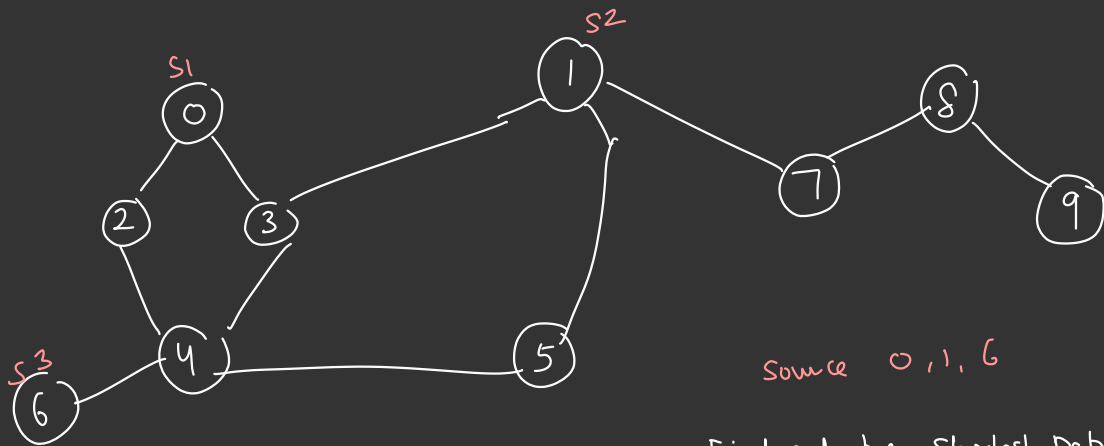


$$\Rightarrow TC = O(N^2) \quad O(N^2 + N^2)$$

$$SC = O(N^2)$$

BFS  
DFS

## "Multi-source BFS"



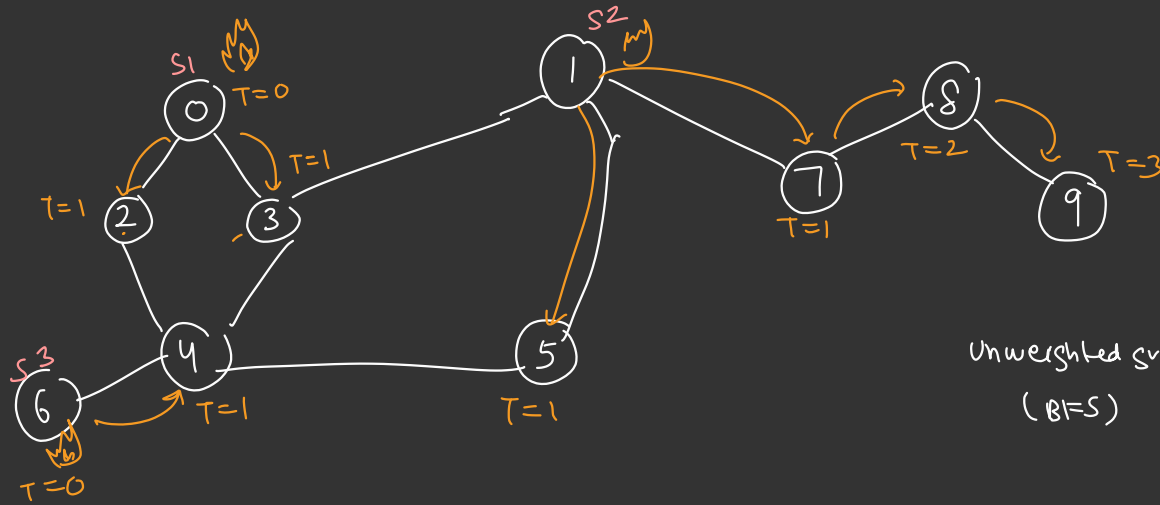
Source 0, 1, 6

Examples

$$d(8) = 2$$

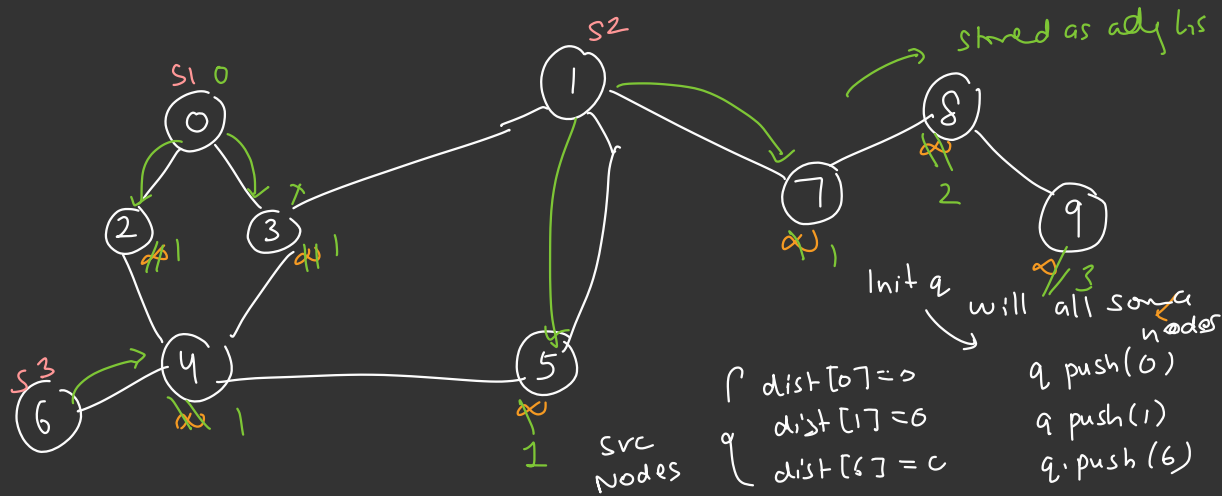
$$d[4] = 1$$

Find out the shortest path len  
for all nodes starting any  
of the source nodes



Unweighted graph  
( $B=5$ )

$T=1$   
 $T=1$   
 $T=3$



$\begin{cases} \text{dist}[0] = 0 \\ \text{dist}[1] = 0 \\ \text{dist}[6] = 0 \end{cases}$

Init q will all source nodes

q.push(0)  
 q.push(1)  
 q.push(6)

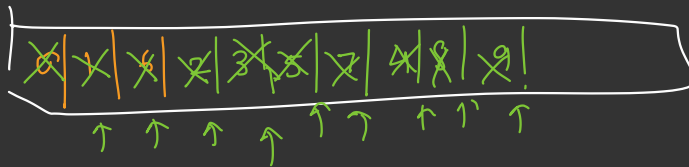
BFS logic  
 while( > 0 )

for( -nbus- )

}

dist[n] == 0

↓  
 not  
 visited.



## Rotten Oranges

1	0	1	2	1
1	1	1	1	1
0	2	0	1	0
0	1	1	1	1
1	1	1	2	0

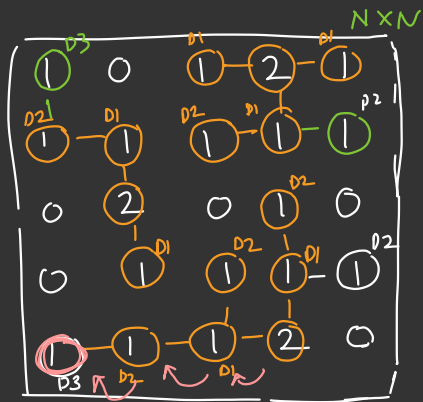
0 - empty

1 - fresh orange

2 - rotten orange



Everyday any fresh orange adjacent to Rotten orange becomes rotten.  $\rightarrow$  4 way  
Find min days in which all oranges will become rotten.

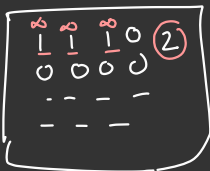


ans = max of all days

↓  
3

↓  
dist of all oranges  
from nearest rotten  
orange  
[Multi-source BFS]

Take  
the  
largest  
dist of FO



Day = ∞

CODE int dist[M][N] = {∞, ∞, ...} → 0 if mat[i][j] = 0

queue < Pair<int, int> > q;

// Init q with pos of Rotten Oranges

```
for(i=0 — M-1) {
    for(j=0 — N-1) {
        if (mat[i][j] == 2) {
            q.push(pair(i, j))
            dist[i][j] = 0
        }
    }
}
```

// BFS - spread in 4 dir

while(!q.empty()) {

i, j = q.poll()

dx = {0, 1, 0, -1}

dy = {1, 0, -1, 0}

for(k=0, k<4; k++) {

ni = i + dx[k]

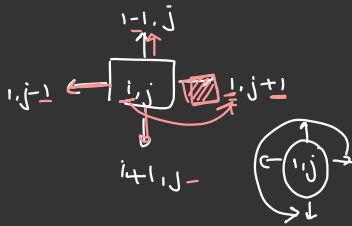
nj = j + dy[k]

each cell goes into Q at max once



$4N^2$

$= O(N^2)$

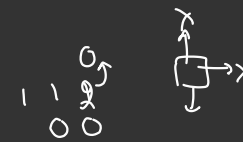
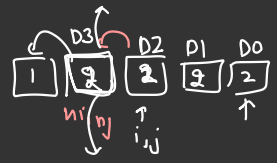


ni, nj

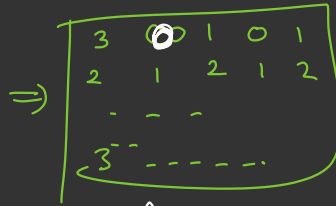
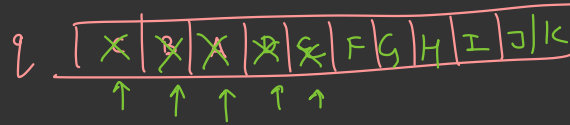
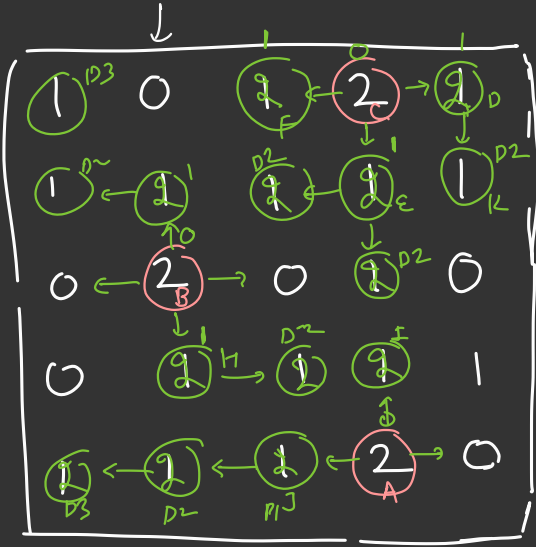
i+0, j+1
i-1, j
i+0, j-1

if (mat[ni][nj] == 0 or outside grid) [ i+1, j ]  
 continue;  
 if (mat[ni][nj] == 1) {  
     ⇒ mat[ni][nj] = 2 ← visited  
     dist[ni][nj] = 1 + dist[i][j]  
     q.push(pair(ni, nj))  
 }  
 }  
 }  
 }

not visited if orange is fresh

for (d : dist) {  
 max-dist = Track the max val of d  
 }  
 return max-dist  
 }



↑  
largest in dist mat ~~1000000~~